Auto-tuning for Memory Hierarchy Optimizations in GPUs

Malik Khan, Gabe Rudy, Chun Chen, Mary Hall
{mmurtaza,grudy,chunchen,mhall}@cs.utah.edu
Jacqueline Chame (jchame@isi.edu)

Funded by NSF award CSR-0615412, and DOE grant DE-FC02-06ER25765.

Motivation & Initial Results

Motivation

- **Challenges of GPU Programming**
  - Performance sensitivity to subtle code differences
  - Complex mapping of indices (threadIdx.x, blockIdx.x)
  - Different programming model from sequential and parallel multi-core code (CUDA)
- **Solution: Automatic Code Generation and Auto-tuning System**
  - Applies common compiler transformations.
  - Finds optimal computation mapping heuristics.
  - Searches highly-optimized code for a target GPU
- **Goal: Achieve performance comparable to manually tuned code**
- **Impact: Real world applications tuned for GPU execution**

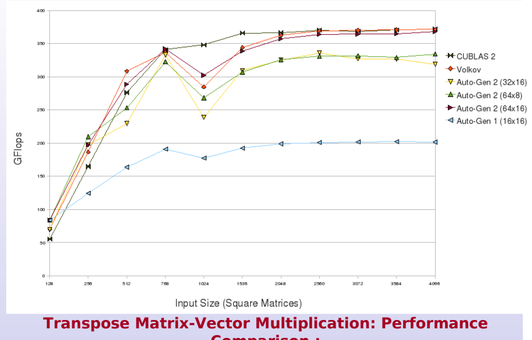
Applying CHiLL Transformations to GPU Code Generation

CHiLL Compiler transformations	Application	
	Conventional Architectures	GPU
Tiling	Manage reuse in limited storage	•Manage reuse in limited storage •Partition parallel execution at 2 levels
Data-copy	Eliminate conflict misses in cache	•Copy data to specialized memory structures
Permutation	Reorder loop structure to enable other optimizations	•Reorder loop structure to enable other optimizations •Optimize data access order
Unrolling	Expose fine-grain parallelism	•Expose fine-grain parallelism •Reduce branch overhead

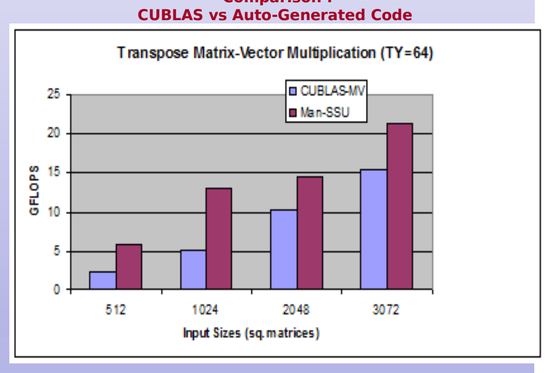
CUDA-CHiLL

- Automatically generate CUDA for NVIDIA GPU from sequential code plus script
- Abstraction permits parallel threads & staging of data
- **Heterogeneous code generation:** Alternative scripts generate CUDA, OpenMP or sequential code tuned for memory hierarchy

Matrix-Matrix Multiplication: Performance Comparison : CUBLAS vs Auto-Generated Code



Transpose Matrix-Vector Multiplication: Performance Comparison : CUBLAS vs Auto-Generated Code



Transformation Recipes

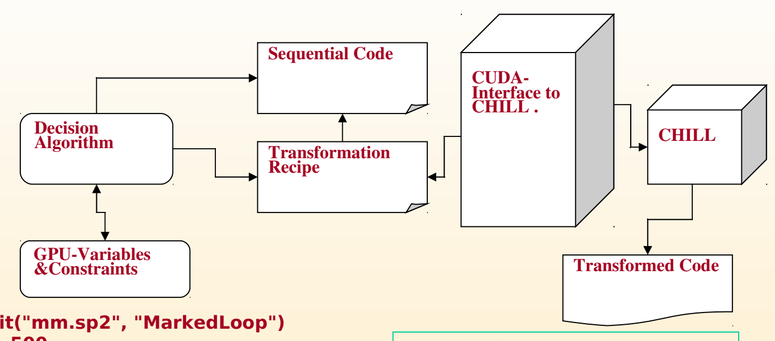
Transformation Scripts

- Separate from code to allow for reuse, sharing and auto-tuning
- Specify a starting nested loop by its logical code position or by a tagging pragma
- Can use references or explicit naming to keep track off and operate on new code resulting from transformations
- Range of abstraction levels
  - High level builds on lower levels (smart shorthand)
- Sound like programming?
  - It is, and it's built on a full-blown language: Lua

Low Level Transformations

- Full power of CHiLL, a polyhedral transformation framework
- Always guarantees correct code at each step
- Adds CUDA transformations to map loops to dimensions of the thread space (grid.{x,y}, thread.{x,y,z})
- Ways of specifying mappings of data to memory space: (global, constant, texture, shared)
- Lifts the "cudaized" computation into a kernel, performing all data copy in and out (based on data dependence analysis) in code generation

CUDA-CHILL Transformation System




```
init("mm.sp2", "MarkedLoop")
N=500
TI=64
TJ=16
TK=TJ

tile_control({"i","j"}, {TI,TJ},
        {l1_control="ii", l2_control="jj"},
        {"ii", "jj", "i", "j"})
tile_control({"k"}, {TK}, {l1_control="kk"},
        {"ii", "jj", "kk","i", "j","k"}, strided)
tile_control({"i"}, {TJ},
        {l1_control="ty",l1_tile="tx"},
        {"ii", "jj", "kk","tx","ty","j","k"})
 --Assign loop levels to thread space and name the kernel
 cudaize("mm_GPU",
       {a=N*N, b=N*N, c=N*N},--array sizes for data copying
       {block={"ii","jj"}, thread={"tx","ty"}})
 --Copy the "c" array usage to registers
 copy_to_registers("kk", "c", {"tx","ty"})
 copy_to_shared("ty", "b")
 --Unroll two innermost loop levels fully
 unroll_to_depth(2)
```

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      c[j][i] += a[k][i]*b[j]
[k];
```

High Level Transformations

- Reason about total loop structure and order
- Can intermix with low-level
- Ensures CUDA mappings and correctness
  - `copy_to_registers` – Privatized data copy to thread registers at the specified loop level, performing extra tiling to map the copy to thread dimensions, adds thread synchronization where appropriate.
  - `copy_to_shared`- Data copy to shared memory, marking the local data as `__shared__`. Performs extra tiling on the copy in and out of shared code to create threads for data staging and adds thread synchronization.

Automatically-Generated Code

```
float P1[16];
__shared__ float P2[16][17];
bx = blockIdx.x, by = blockIdx.y;
tx = threadIdx.x, ty = threadIdx.y;
P1[0:15] = c[16*by:16*by+15][tx+64*bx+16*ty];
for (t6 = 0; t10 <= 1008; t6+=16) {
  P2[tx][4*ty:4*ty+3] = b[16*by+4*ty:16*by+4*ty+3][tx+t6];
  __syncthreads();
  P1[0:15] += a[t6][64*bx+16*ty+tx]*P2[0][0:15]
  P1[0:15] += a[t6+1][64*bx+16*ty+tx]*P2[1][0:15]
  ...
  P1[0:15] += a[t6+15][64*bx+16*ty+tx]*P2[15][0:15]
  __syncthreads();
}
c[16*by:16*by+15][tx+64*bx+16*ty] = P1[0:15];
```

Decision Algorithm

- Decision algorithm: Automatically derive transformation recipes for specific code structures.
  - Derive transformations and compositions for specific computations.
  - Perform tiling transformation to decompose computation into thread and block dimensions
    - Partition to ***hierarchically tiled architecture***).
  - Manage ***heterogeneous memory hierarchy***
    - Select memory structure (global, shared, constant, texture memory or registers)
    - Explicit data copy from global memory into shared memory or registers.
  - Uniquely employs and adapts CHiLL transformations for GPUs
    - Changes to tiling and data copy implementations.
  - Interact with auto-tuning framework to generate code variants and search the space of possible variants and parameter values.
    - Compiler can take risks on optimization strategies and rely on auto-tuning to rule out unprofitable solutions.

Auto-Tuning Applications

- Auto-tuning of real world applications and kernels (current and future work)
- Applications
  - MADNESS (ORNL)
  - quantum mechanical code (USC)
- Libraries
  - CUBLAS examples
  - PETSc
  - Domain-specific library for partial differential equation solutions
- Stencils and other computations
  - Jacobi relaxation
  - Sobel edge detection
- Benchmarks
  - High CUDA benchmarks
  - CP, MM4K, MRI-FHD
  - NAS OpenMP parallel benchmark examples

Contributions

- Higher level abstraction for expressing CUDA code generation from sequential code.
- Decision algorithm to derive transformation recipes for specific domains, adapted to the GPU architecture.
- Auto-tuning applied to real world applications, libraries & kernels using compiler framework.

Impact:

- Increase programmer productivity in developing GPU libraries and applications.
- ***Achieve performance comparable and sometimes better than CUBLAS library.***
- A step towards supporting portable heterogeneous applications targeting a variety of architectures.

Future Work:

- OpenMP and OpenCL code generation